$$\sum_{t \in trans-trans(i)} \max_{j \in hp(i) \cap tasks(t)} \left(I_t(\hat{O}_j) \right) \tag{18}$$

The interference from the transaction of which i is a member is given by:

$$I_{trans(i)}(W) \tag{19}$$

Where W is the start of the window, measured relative to the start of the transaction. Recall that we define the transaction *trans(i)* such that task i has an offset $\geq O_j$ (where j is any other task in the same transaction as i). Hence $W \leq O_i$.

For a given W, then, the total computation that takes place in a window of width w (and hence the window width w) is equal to the computation of i itself, plus the computation of tasks in the same transaction as i, and the computation of tasks in other transactions:

$$w = C_i + I_{trans(i)}(W) + \sum_{t \in trans-trans(i)} \max_{j \in hp(i) \cap tasks(t)} \left(I_t(\hat{O}_j) \right) \tag{20}$$

For a given W and w, the response time of a task i is equal to $W + w - O_i$ (see equation 10). The worst-case response time for a task i is therefore given by:

$$r_i = \max_{W \in \hat{O}_j \cdot \forall j \in trans(i) \cap (hp(i) \cup \{i\})} \left(W + w - O_i \right) \tag{21}$$

Compare the above equation with equation 16; the above equation is computationally much easier to evaluate, being $O(nm)$ instead of $O(n^m)$. However, before adopting the test a comparison must be made of the relative performances of the two equations for a range of task sets.

Comparison of Exact and Tractable Test Relative Performance

We propose to evaluate the performance of the tests described by equations 21 and 16, and the original test, equation 4. To do this we constructed a number of random task sets for varying processor utilisations, and evaluated (using all three tests) the worst-case response time for the lowest priority task in the set (denoted as task i). The performance index of each test was defined as:

$$\frac{r_i}{T_i}$$

Obviously smaller values indicate a better result. The random task sets were constructed as follows:

A random[8] number of transactions were chosen; for each transaction a random number of tasks were chosen (to keep the problem tractable for the exact test[9], the number of tasks per transaction was limited to six, and the number of transactions was limited to five). Each transaction was assigned a random period, with a minimum period of 50 and a maximum period of 1000. Each task was assigned a random fraction between 0 and 1, indicating the relative share of processor utilisation. The worst-case execution time of each task was then assigned in proportion to each task's utilisation share such that the total processor utilisation is equal to a randomly chosen utilisation bound. A random priority ordering of all tasks was then chosen. During the evaluation, if the worst-case response time of any test exceeded the period then the worst-case response time was deemed to be equal to the task period (giving a performance index for that task of 1.0).

Ten thousand such task sets were generated, and each test applied to the lowest priority task (Figure 8). In 93% of the task sets the tractable test result was the same as the exact test.

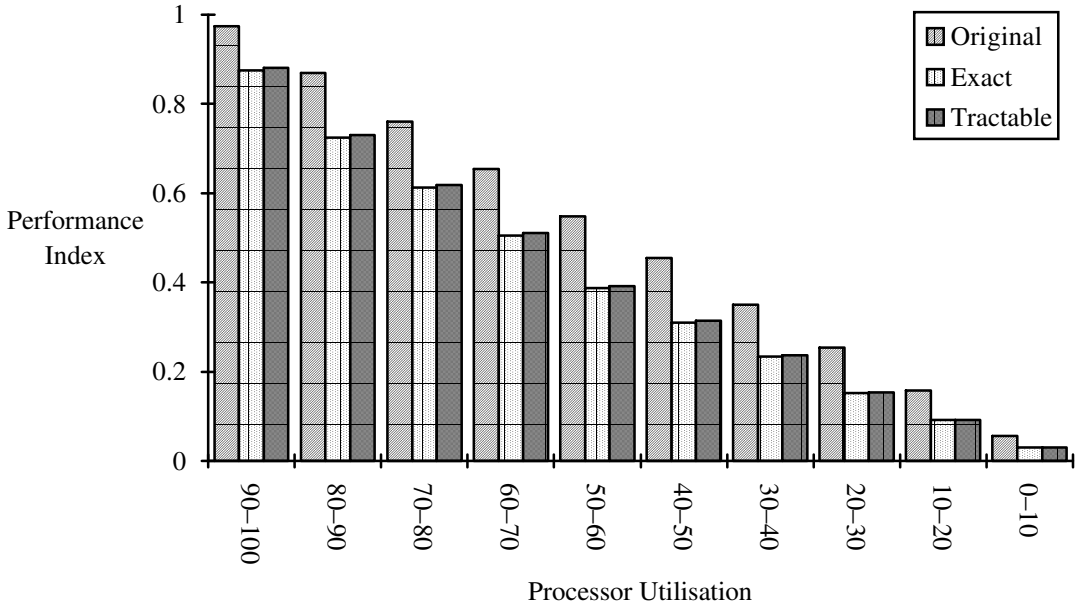

Figure 8: relative performances of three schedulability tests

On average, the Tractable Offset Test took 13.2ms (on a Sun 3/280) to compute the schedulability; the Exact Offset Test took 277ms; the old test took 0.4ms.

5. THE OFFSET TEST AND ACCESS TO SHARED RESOURCES

This section shows how, for certain problems, access to shared resources (such as buffers) can be allowed without using semaphores or any other dynamic concurrency control protocol. Furthermore, we also show how the offset test can be extended to

[8]We use the term 'random' as meaning 'uniform random'

[9]Because of the large amount of computation required to evaluate the exact test a 50MIP RS6000 computer was used

permit the analysis of tasks that do lock and unlock semaphores according to the priority ceiling protocol [13].

Tasks within the same transaction maintain offset relationships; resources shared exclusively between tasks in the same transaction will therefore have a known access pattern. If the time windows over which the tasks access the resource do not overlap then no dynamic concurrency control protocol is needed: for example, consider two tasks *x* and *y*, each in the same transaction. Task *x* and *y* are members of a periodic transaction. Task *x* performs computation on sensor readings (read from a buffer shared with task *y*), and places results back in the buffer. Task *y* reads from a sensor, places the reading in the buffer, and then reads the output of task *x* from the buffer and sends it to an actuator. Task *y* has a tight deadline to meet strict input and output jitter requirements. Task *x* has a much longer deadline since it must produce the results of its computation by a (much longer) lag requirement (Figure 9). Task *y* is assigned a relatively high priority to ensure that the jitter requirements are met.

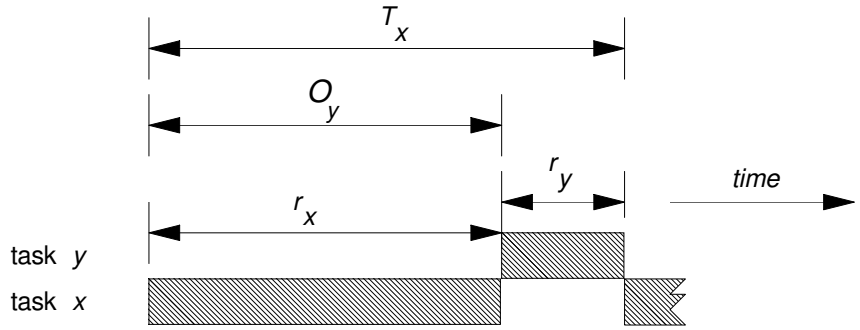

Figure 9: access to a shared buffer by two tasks, *x* and *y*

The lag time of the example is equal to the period of the transaction. The input and output jitter of the example is equal to r_y.

As long as both tasks cannot hold the resource concurrently there is no need to guard the shared buffer with a semaphore: this can be generalised to any number of tasks in the same transaction sharing a resource by considering just two tasks *a* and *b*, where task *b* is of higher priority than *a*.

If task *b* is released when task *a* is active (*i.e.* task *a* has already been released) then task *b* will always pre-empt task *a*, and hence no concurrent access to a resource shared between tasks *a* and *b* can occur (of course, task *b* must complete before the next invocation of task *a*). If task *b* is released before *a* then it must complete before task *a* starts.

No concurrent access to a resource shared between two tasks *a* and *b* can occur if the following condition holds

$$\left(O_a \geq O_b \wedge O_a + r_a \leq O_b + T\right) \vee \left(O_a + r_a \leq O_b\right) \tag{22}$$

For resources shared between tasks in the same transaction where the above conditions are unable to be met, or for resources shared between tasks of different transactions, a

concurrency control protocol, such as the priority ceiling protocol, needs to be used. The priority ceiling protocol guarantees that any given task i can be blocked (or deferred by so-called 'push through blocking') at most once by a lower priority task accessing a semaphore with a ceiling of equal or higher priority than task i. Thus a blocking factor B_i can be assigned to i, denoting the worst-case time task i could be blocked.

Equation 4 shows how the blocking factor is incorporated into the original schedulability test — the window (of width r_i) contains computation from higher priority tasks, from task i, and from a lower priority task for the time it inherits a priority higher than or equal to the priority of task i.

The blocking factor in the Tractable Offset Test can be no more than the blocking factor determined by the original analysis[10] [5]. However, it may be possible to determine a lower bound than this, since lower priority tasks in the same transaction as i are released at fixed offsets in relation to i — a lower priority task released immediately after i cannot block that invocation of i (since i will prevent the lower priority task from being run and hence from locking a semaphore). Now, recall that we define $\hat{O}_j$ for any task j in the transaction $trans(i)$ such that $\hat{O}_j < O_i$ (equation 5). If a lower priority task k always terminates before the start of the window (the window starts at time W) then k can never block i directly. Furthermore, no other task in the transaction starting after W can be blocked by k (and then indirectly block task i). Any instance of a task of higher priority than i released before time W is not counted in the computation of w. Hence, a lower priority task k is not included in the calculation of the blocking factor B_i (and hence does not contribute to the length of the 'level i busy period' starting at time W) if the following condition holds:

$$\left(O_k \geq W \wedge O_k + r_k \leq W + T\right) \vee \left(O_k + r_k \leq W\right)$$

Using the re-defined offset $\hat{O}$ we can rewrite the above as:

$$\hat{O}_k + r_k \leq W \tag{23}$$

Note, however, that the above is a sufficient but not exact representation of the calculation of the worst-case blocking time: it could be that the task release pattern that causes the worst-case response time for task k is not the same pattern that causes the worst-case response time for task i, and thus any delay suffered by i due to blocking by k might be compensated for by a smaller interference on task i.

We can extend the offset test by rewriting equation 20:

$$w = C_i + B_i(W) + I_{trans(i)}(W) + \sum_{t \in trans - trans(i)} \max_{j \in hp(i) \cap tasks(t)} \left(I_t(\hat{O}_j)\right) \tag{24}$$

[10] "longest critical section of any task with lower priority than task i accessing a semaphore with ceiling priority equal or higher than the priority of task i."

Where $B(W)$ is a function giving the blocking factor for task i when considering a window starting at time W, and is calculated as in the original protocol definition [13], except for the contribution from as lower priority task k in the same transaction as i: such a task k does not contribute to the calculation of the blocking factor if the condition given by equation 26 holds. Note that this condition requires the worst-case response time r_k to be found before finding the worst-case response time r_i, and thus an order for evaluating worst-case response times is imposed — the worst-case response time for the lowest priority task must be found first.

6. DISCUSSION

So far we have shown how periodic (or sporadic) transactions of tasks, with time offsets between tasks of the same transaction, can be analysed using newly derived scheduling theory. This theory is able to find good worst-case response time bounds. Furthermore, we have shown how time offsets are particularly useful in expressing some common real-time requirements. We then discussed how the priority ceiling protocol can be added to the offset test analysis, and discussed how offsets could be used to provide an alternate concurrency control mechanism. We now discuss some of the ramifications of the offset scheduling theory.

Unrelated strictly periodic tasks sharing the same period can be incorporated into the same transaction, with offsets between the tasks. This increases schedulability because the computation due to these tasks can be 'spread out'. Tasks with similar but different periods can be transformed into tasks sharing the same period by choosing a common period which is smaller than the original periods. This has the advantage of reducing the least common multiple of the task periods, and simplifying the transaction (at the expense of a small loss in schedulability). Tasks with periods which are exact divisors of a given transaction period can also be incorporated into the transaction by adding multiple instances of the same task with offsets between them. The offset approach is more general that this, however, since it allows any instance to have different attributes (*e.g.* priority); this generality can be used to improve schedulability (for example, if only one instance of a task were unschedulable then the priority of just that instance might be increased). Section 11 illustrates the construction of such a transaction with an extended example derived originally by Audsley [4]. Tasks in the same transaction which share resources can be configured such that they never concurrently access the resource, and hence semaphores need not be used to guard access to the resource. This is useful since it removes complexity from any run-time support software, and also reduces shared resource access overheads.

The above description of constructing transactions will be familiar: it is exactly the procedure that is adopted in finding static cyclic schedules [18]. It therefore reasonable to conclude that the Tractable Offset Test, coupled with a method for configuring tasks (choosing priorities [4], offsets, *etc.*), provides a means by which static cyclic schedules can be analysed. However, the offset approach provides a major additional benefit: other transactions are also permitted to run concurrently. Thus a number of sporadic and periodic tasks can be run alongside any cyclic schedule. Indeed, a number of cyclic schedules can be run concurrently on the same processor. Thus we can see that the Tractable Offset Test bridges the gap between static cyclic scheduling and

static priority pre-emptive scheduling, and that static cyclic scheduling is merely a special case of the more general pre-emptive scheduling algorithm.

We can illustrate this by using an example derived by Xu and Parnas [19]. They state that:

"*Static priority-driven schemes are only capable of producing a very limited subset of the possible schedules for a given set of processes. This severely restricts the capability of priority-driven schemes to satisfy timing and resource sharing constraints at run-time*"

They then go on to say:

"*An example is shown* [see below] *where it is assumed that process segment B is not allowed to pre-empt process segment A. The timing constraints then require that the processor must be left idle between time 0 and 11, even though A's release time is 0; if A starts it would cause B to miss its deadline. Note that in this example, some segments are pre-emptable, but others are not. In most real situations, there will be process segments that cannot be pre-empted because a critical section has been entered. Thus this example illustrates a very common situation.* ***Neither static nor 'dynamic' priority-driven schemes can deal properly with such situations***."

We believe this assertion is incorrect — if the implementation of the pre-emptive priority scheduling supports time offsets then any system schedulable by cyclic executives can be scheduled (in the worst-case we can always transform the task set to have identical execution patterns to the static schedule!). Furthermore, we will show that the example can be both scheduled and analysed by giving the example and then applying the offset test to it. Firstly we indicate how precedence and exclusion constraints can be met using a pre-emptive static priority dispatcher and offsets. We will then take an example derived by Xu and Parnas (which is claimed can only be scheduled using a static cyclic schedule) and show how the Tractable Offset Test can schedule the task set, and how the precedence and exclusion constraints are met.

Meeting Precedence and Exclusion Constraints

This paper has already said much about exclusion constraints, using both the priority ceiling protocol to guard small critical sections, and by using offsets and priority assignment to ensure total exclusion[11]. We have yet to adequately address precedence constraints.

Constraint: *A task B is constrained to run only when a task A has finished*

This constraint can be achieved two ways: through offsets and through priority. We deal with the priority approach first. If task *B* is assigned a lower priority than task *A* and is released at the same time as (or after) task *A* then task *A* will immediately pre-

[11]Xu and Parnas use the term 'exclusion' between two tasks to mean no part of the execution of either task can overlap; where the critical section is small a total exclusion constraint is too harsh, and an alternative concurrency control protocol (such as the priority ceiling protocol) can be used more efficiently; of course, the priority ceiling protocol cannot be used in a static cyclic schedule

empt B (or already be released and delay B). Task B will not be dispatched until it becomes the highest priority task; this cannot happen until task A has finished. Therefore we can say that the precedence constraint will be satisfied if:

$$\left(O_B \geq O_A \wedge O_B + r_B \leq O_A + T\right) \wedge \text{pri}(B) < \text{pri}(A) \tag{25}$$

Given that task B is of lower priority than task A.

The following approach can be used when task B has a lower *or* higher priority than task A: if task B has an offset larger than the offset of A *plus* the worst-case response time of A then task B will never be released before A is finished, *i.e.*:

$$\left(O_B \geq O_A + r_A\right) \wedge \left(O_B < O_A\right) \tag{26}$$

We have to be careful about periodic task execution where the subsequent invocation of a task may disturb the precedence relationship. For example, if task A were assigned a priority higher than task B then the subsequent re-arrival of task A could pre-empt a currently running invocation of B. If we additionally require exclusion between B and the subsequent invocation of A then we must ensure that B always finishes before A is re-released. Alternatively, we could turn task A into two logically different tasks, each of period twice that of the original, with an offset equal to the original period between the two tasks. We could then assign a lower priority to the second task A so that task B would not be pre-empted. In the example we take precedence to be stronger than exclusion and hence require that task B terminate before the subsequent invocation of task A.

Example Task Set

Five tasks, *A-E,* all share the same period, have exclusion and precedence relationships:

1. B precedes D
2. A excludes D
3. A excludes B.

The following table gives the requirements, as stated by Xu and Parnas:

	d	r_{Xu}	C
A	161	0	30
B	51	11	30
C	90	60	30
D	100	41	10
E	140	90	50

C is the worst-case execution time; d is the deadline of the task, measured from time zero; r_{Xu} is the minimum release time measured from time zero.

Note that we have chosen a value of 30 for the worst-case execution time of task *C* — Xu and Parnas state that the worst-case execution time is 10, but the time diagram they give inconsistently uses 30; for compatibility with the diagram we will assume the more challenging value of 30.

Now consider the modified task set we can obtain:

	d	r_{Xu}	*C*	*D*	*O*	*pri*
A	161	0	30	110	51	5
B	51	11	30	40	11	1
C	90	60	30	30	60	3
D	100	41	10	59	41	2
E	140	90	50	50	90	4

The timing constraint *d* stated by Xu and Parnas is measured relative to time zero. In our offset analysis we measure the deadline from the release time of the task. Therefore we obtain a new deadline *D* measured from the release of the task and equal to $d - O$. Furthermore, r_{Xu} is the minimum release time of the task; *O* must be greater than this required value.

We have set the offset of task *A* such that task *A* cannot not execute before time 51. Furthermore, we have assigned static priorities[12] to the five tasks (Audsley [4] gives an algorithm for finding the optimal priority ordering for such task sets) — we intend the tasks to be scheduled according to the static priority pre-emptive scheduling algorithm. Note that in this example there are only five tasks, all members of the same transaction — using a pre-emptive scheduler in a larger system we could have other sporadic and periodic tasks (using the static cyclic scheduling approach it is very difficult to guarantee tight bounds on servicing sporadic requests).

Running the modified task set through the offset test tool we obtain:

	d	r_{Xu}	*C*	*D*	*O*	*pri*	*r*	r_{old}	
A	161	0	30	110	51	5	110	150	✓
B	51	11	30	40	11	1	30	30	✓
C	90	60	30	30	60	3	30	70	✓
D	100	41	10	59	41	2	10	40	✓
E	140	90	50	50	90	4	50	120	✓

Where *r* is the worst-case response time of the task, measured relative to *O*, the offset (which is itself measured relative to time zero). The column headed r_{old} is the old scheduling analysis applied to the same problem; we can see how poor the performance of the old analysis is in this situation. The ✓ symbol indicates that the timing constraint ($r \leq D$) has been met.

Meeting Constraints: The Example

[12] 1 denotes the highest priorty, 5 the lowest

Recall that we have three constraints that must be met:

1. task *B* finishes before task *D* starts executing
2. task *A* cannot be pre-empted by task *D* (and *vice versa*)
3. task *A* cannot be pre-empted by task *B* (and *vice versa*)

We can see that constraint (1) is met — the following condition is met (equation 26):

$$(O_D \geq O_B + r_B) \wedge (O_D < O_B + T)$$

Task *B* is released at time 11 and will take no longer than 30, and hence will finish by time 41. Task *D* is released at time 41 and hence task *D* executes only after task *B* has finished. Note that task *D* is assigned a lower priority than task *B* and could therefore have been released earlier (task *B* would have pre-empted task *D*), as long as the following condition is met (equation 25):

$$(O_D \geq O_B \wedge O_D + r_D \leq O_B + T) \wedge \text{pri}(D) < \text{pri}(B)$$

Constraints (2) and (3) are also met — for exclusion between tasks *A* and *D* we have (equation 25):

$$\begin{aligned}
&(O_A \geq O_D \wedge O_A + r_A \leq O_D + T) \vee (O_A + r_A \leq O_D) \\
&= (51 \geq 41 \wedge 51 + 110 \leq 41 + 161) \vee (51 + 110 \leq 41) \\
&= (51 \geq 41 \wedge 51 + 110 \leq 41 + 161) \\
&= \mathit{true}
\end{aligned}$$

Similarly for *A* and *B* we have:

$$\begin{aligned}
&(O_A \geq O_B \wedge O_A + r_A \leq O_B + T) \vee (O_A + r_A \leq O_B) \\
&= (51 \geq 11 \wedge 51 + 110 \leq 11 + 161) \vee (51 + 110 \leq 11) \\
&= (51 \geq 11 \wedge 51 + 110 \leq 11 + 161) \\
&= \mathit{true}
\end{aligned}$$

Hence we can see that the timing, precedence, and exclusion constraints will be met. As a final arbiter, we will use the *STRESS* simulator to execute the five tasks using static priority pre-emptive scheduling. The following output was produced by this program:

7. SUMMARY AND CONCLUSIONS

This section summarises the Tractable Offset Test developed in this paper. For a given task i, member of a transaction denoted *trans(i)*, the worst-case response time of that task i, denoted r_i, is given by:

$$r_i = \max_{W \in \hat{O}_j \cdot \forall j \in trans(i)} \left(W + w - O_i \right)$$

Where O_i is the offset of task i relative to the start of the transaction *trans(i)*, and w is found iteratively by:

$$w^{n+1} = C_i + B_i(W) + I_{trans(i)}(W) + \sum_{t \in trans - trans(i)} \max_{j \in hp(i) \cap tasks(t)} \left(I_t(\hat{O}_j) \right)$$

Where $w^0 = C_i$, C_i is the worst-case computation time required by task i, T_j is the period of task j, $hp(i)$ is the set of all tasks of higher priority than i, *trans* is the set of all transactions on the processor, *trans*(i) is the transaction of which task i is a member, *tasks*(t) is the set of tasks that are members of a transaction t. $I_t(x)$ is given by:

$$I_t(x) = \sum_{k \in hp(i) \cap tasks(t)} \left(\left\lceil \frac{x + w^n - \hat{O}_k}{T_k} \right\rceil_0 - \left[x - \hat{O}_k \right]_0^1 \right) C_k$$

And $\hat{O}_j$ is given by:

$$\hat{O}_j = \begin{cases} O_j & \text{if } O_j < O_i \vee j \notin tasks(trans(i)) \\ O_j - T_j & \text{otherwise} \end{cases}$$

$B_i(W)$ is an upper bound on the time task i can be blocked. It is bounded by the longest critical section of a semaphore with ceiling priority greater than or equal to the priority of task i that is locked by any relevant lower priority task. A relevant task a is one which belongs to a different transaction, or if the following condition holds:

$$\hat{O}_a + r_a > W$$

We have seen in this paper how ‘transactions of tasks’ is a useful way of structuring hard real-time software, and how offset information between tasks can be used to obtain good bounds on worst-case response times for such tasks. We have also seen how offsets between tasks can sometimes allow the removal of semaphores guarding certain shared resources. Furthermore, we have also seen how such a task model encompasses task sets scheduled according to a pre-computed static cycle; we hope that this result will lead to increased investigation of the configuration and analysis of static priority pre-emptively scheduled hard real time systems.

8. ACKNOWLEDGEMENTS

The author would like to thank Professor J. A. McDermid, Dr A. Burns, and Dr A. J. Wellings for their valuable comments on earlier drafts of this paper.

9. REFERENCES

[1] Audsley, N. C., "*Deadline Monotonic Scheduling,*" Computer Science Department, University of York (September 1990).

[2] Audsley, N. C., A. Burns, M. F. Richardson and A. J. Wellings, "*Hard Real-Time Scheduling: The Deadline Monotonic Approach,*" Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software (15-17 May 1991).

[3] Audsley, N. C., A. Burns, M. F. Richardson, A. J. Wellings, "*STRESS: A Simulator For Hard Real-Time System*", Real-Time Systems Research Group Report RTRG/91/106, Department of Computer Science, University of York, October 1991

[4] Audsley, N. C., "*Optimal Priority Assignment and Feasibility of Static Priority Tasks With Arbitrary Start Times,*" Dept. Computer Science, University of York (December 1991).

[5] Audsley, N. C., A. Burns, M. Richardson, K. Tindell, A. Wellings, "*Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling*", Report number RTRG/92/120, Feb 1992

[6] Joseph, M., P. Pandya, "*Finding Response Times in a Real-Time System,*" BCS Computer Journal (Vol. 29, No. 5, Oct 86) pp.390-395.

[7] Kopetz, H., W. Ochsenreiter, "*Clock Synchronisation in Distributed Real-Time Systems,*" IEEE Transactions on Computers C-36(**8**) (August 1987).

[8] Lehoczky, J., L. Sha, Y. Ding, "*The Rate-Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behaviour"*, Proceedings IEEE Real-Time Systems Symposium, pp166-171, December 1989

[9] Lehoczky, J. P., "*Fixed Priority Scheduling of Periodic Task Sets With Arbitrary Deadlines,*" Proceedings 11th IEEE Real-Time Systems Symposium (5-7 Decmeber 1990) pp.201-209.

[10] Leung, J. Y. T., J. Whitehead, "*On The Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks,*" Performance Evaluation (Vol. 2, Part 4, Dec 1982) pp.237-250.

[11] Liu, C. L., J. W. Layland, "*Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment,*" Journal of the ACM 20(**1**) (1973) pp.46-61.

[12] Locke, C.D, "*Software architecture for hard real-time applications: cyclic executives vs.fixed priority executives*", Real-Time Systems 1(**4**) (March 1992) pp.37-53

[13] Sha, L., R. Rajkumar and J. P. Lehoczky, "*Priority Inheritance Protocols: An Approach to Real-Time Synchronisation,*" IEEE Transactions on Computers 39(**9**) (September 1990) pp.1175-1185.

[14] Simpson, H., "*Four-Slot Fully Asynchronous Communication Mechanism,*" IEE Proceedings (Vol. 137, Pt. E., No. 1, Jan 1990) pp.17-30.

[15] Strosnider, J. K., T. E. Marchok, "*Responsive, Deterministic IEEE 802.5 Token Ring Scheduling*", Journal of Real-Time Systems 2(**1**) (September 1989) pp.133-158

[16] Tindell, K. W., A. Burns and A. Wellings, "*Allocating Real-Time Tasks (An NP-Hard Problem made Easy),*" Real Time Systems (To Appear).

[17] Tindell, K. W., A. Burns, A. J. Wellings, "*Guaranteeing Hard Real Time End-to-End Communications Deadlines*", Report number RTRG/91/107, Dec 1991

[18] Xu, J., D. L. Parnas, "*Scheduling Processrs with Release Times, Deadlines, Precedence, and Exclusion Relations,*" IEEE Transactions on Software Engineering 16(**3**) (March 1990) pp.360.

[19] Xu, J., D. L. Parnas, "*On Satisfying Timing Constraints in Hard-Real-Time Systems,*" ACM Software Engineering Notes 16(**5**) (December 1991) pp.132-146.

APPENDIX A: CONVERGENCE OF ORIGINAL ANALYSIS

The original analysis is guaranteed to converge if either $r_i^{n+1} = r_i^n$, or r_i^{n+1} exceeds some threshold (such as the task deadline). To prove, therefore, that the iteration to find r_i terminates we need only show that successive approximations to r_i are monotonically increasing. We prove this for a task i with a single higher priority task j; the extension to many higher priority tasks is trivial.

Theorem

Given:

$$r_i^{n+1} = C_i + \left\lceil \frac{r_i^n}{T_j} \right\rceil C_j$$

Where C_i, C_j, and T_j are integer constants ≥ 0, and $r_i^0 = 0$

Then:

$$r_i^{n+1} \geq r_i^n$$

Lemma

$$r_i^n \geq r_i^{n-1} \rightarrow r_i^{n+1} \geq r_i^n$$

Proof

If:

$$r_i^n \geq r_i^{n-1}$$

Then:

$$\frac{r_i^n}{T_j} \geq \frac{r_i^{n-1}}{T_j}$$

Therefore:

$$\left\lceil \frac{r_i^n}{T_j} \right\rceil \geq \left\lceil \frac{r_i^{n-1}}{T_j} \right\rceil$$

Hence:

$$C_i + \left\lceil \frac{r_i^n}{T_j} \right\rceil C_j \geq C_i + \left\lceil \frac{r_i^{n-1}}{T_j} \right\rceil C_j$$

And so:

$$r_i^{n+1} \geq r_i^n$$

□

Proof of Theorem

We have:

$$r_i^0 = 0$$

and:

$$r_i^1 = C_i$$

Therefore $r_i^1 \geq r_i^0$, and hence by induction $r_i^{n+1} \geq r_i^n$.

□

In a similar manner it is possible to prove that the equation is guaranteed to converge to a value if the processor utilisation is less than 100% (the final value of r_i may, of course, exceed T_i).

APPENDIX B: EQUIVALENCE TO ORIGINAL ANALYSIS

This section will take the derived offset test and show how it is equivalent to the original analysis when all tasks within a transaction share the same offset. Further, we will also show how the original analysis for so-called 'sporadically periodic' tasks described by Audsley *et al* can be alternatively derived from the offset test.

We can remove offset information from the Tractable Offset Test by setting all offset values to zero, giving the following:

$$w = C_i + B_i + I_{trans(i)}(0) + \sum_{t \in trans - trans(i)} \max_{j \in hp(i) \cap tasks(t)} \left(I_t(0) \right) \qquad (27)$$

$$I_t(0) = \sum_{j \in hp(i) \cap tasks(t)} \left(\left\lceil \frac{w}{T_j} \right\rceil_0 - [0]_0^1 \right) C_j$$

I_t can be re-written:

$$I_t = \sum_{j \in hp(i) \cap tasks(t)} \left\lceil \frac{w}{T_j} \right\rceil C_j$$

So:

$$w = C_i + B_i + I_{trans(i)}(0) + \sum_{t \in trans - trans(i)} I_t(0)$$

$$= C_i + B_i + \sum_{t \in trans} I_t(0)$$

$$= C_i + B_i + \sum_{t \in trans} \sum_{j \in hp(i) \cap tasks(t)} \left\lceil \frac{w}{T_j} \right\rceil C_j$$

$$= C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{w}{T_j} \right\rceil C_j$$

Now, the worst-case response time of task *i* is given by:

$$r_i = \max_{W \in O_j \cdot \forall j \in trans(i)} \left(W + w - O_i \right)$$

or:

$$r_i = \max(w)$$
$$= w$$

So:

$$r_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{r_i}{T_j} \right\rceil C_j$$

And hence the original analysis has been re-established (Equation 4).
□

We now turn to the reduction of the Tractable Offset Test to the problem of so-called 'sporadically periodic tasks'. Sporadically periodic tasks are tasks which arrive at some

time, execute periodically (termed the 'inner period') for a number of times, and then do not re-arrive for some time (termed the 'outer period'). A sporadically periodic task i is be characterised by:

C_i	worst-case execution time of a task i
n_i	maximum number of periodic executions of task i between outer arrivals
T_i	outer period
t_i	inner period

It was required that $r_i \leq t_i$ for any task i. The original analysis was extended by Audsley *et al* [5] to handle these tasks, and produced the following:

$$r_i = C_i + B_i + \sum_{j \in hp(i)} \left(Kn_j C_j + \left\lceil \frac{r_i - KT_j}{t_j} \right\rceil C_j \right) \tag{28}$$

Where:

$$K = \left\lfloor \frac{r_i}{T_j} \right\rfloor \tag{29}$$

We will show how the Tractable Offset Test can be reduced to the above for sporadically periodic tasks.

A sporadically periodic task i can be modelled as a transaction t of period T_i with n_i tasks each requiring computation C_i, and with offsets of 0, t_i, $2t_i$, ..., $(n\text{-}1)t_i$. We have:

$$w^{n+1} = C_i + B_i(W) + I_{trans(i)}(W) + \sum_{t \in trans-trans(i)} \max_{j \in hp(i) \cap tasks(t)} \left(I_t(O_j) \right)$$

We can rewrite the above equation, replacing O_j with a multiple of t_j. Furthermore, since we assume that $r_i \leq t_i$ we can remove the part of the above equation relating to interference from tasks within the same transaction (*i.e.* all tasks within transaction t suffer no interference from each other), and hence define W to be equal to zero. Finally, $B_i(0)$ is defined to be equal to the blocking factor deemed by the original analysis, B_i — no task in the same transaction as i can contribute to the blocking factor B_i since all such tasks must finish before i is released. Hence:

$$w^{n+1} = C_i + B_i + \sum_{t \in trans-trans(i)} \max_{k=1..n_t} \left(\sum_{j=1..n_t} \left(\left\lceil \frac{kt_t + w^n - (j-1)t_t}{T_t} \right\rceil_0 - \left[kt_t - (j-1)t_t \right]_0^1 \right) C_t \right)$$

We can re-write the above equation thus:

$$w^{n+1} = C_i + B_i + \sum_{t \in trans-trans(i)} \max_{k=1..n_t} \left(\sum_{j=1..n_j} \left(\left\lceil \frac{(k+1-j)t_t + w^n}{T_t} \right\rceil_0 \right) C_t - kC_t \right)$$

The maximum value of the right hand clause occurs when $k = 0$. This can be seen by considering what happens when $k \rightarrow k + 1$. The ceiling function can increase by at most C_t, but the overall clause can only stay the same or decrease. Therefore $k = 0$ will result in the worst-case behaviour. Hence we can rewrite the above equation:

$$w^{n+1} = C_i + B_i + \sum_{t \in trans - trans(i) \wedge pri(t) > pri(i)} \sum_{j=1..n_t} \left\lceil \frac{(1-j)t_t + w^n}{T_t} \right\rceil_0 C_t$$

At some value of j (say α) the term

$$\left\lceil \frac{(1-j)t_t + w^n}{T_t} \right\rceil_0 C_t$$

falls in value from $(K + 1)C_t$ to KC_t, where K is defined by:

$$KT_t < w \leq (K+1)T_t$$

Hence:

$$w - t_t(\alpha - 1) > KT_t \wedge w - t_t\alpha \leq KT_t$$

$$\alpha - 1 < \frac{w - KT_t}{t_t} \leq \alpha$$

Therefore α is given by:

$$\alpha = \left\lceil \frac{w - KT_t}{t_t} \right\rceil C_t$$

and hence the total interference is given by:

$$\alpha C_t + n_t K C_t$$

The full schedulability equation is thus:

$$w = C_i + B_i + \sum_{\forall t \in trans \wedge pri(t) > pri(i)} \left(n_t K + \left\lceil \frac{w - KT_t}{t_t} \right\rceil \right) C_t \qquad (30)$$

Also:

$$r_i = \max_{W \in O_j \cdot \forall j \in trans(i)} \left(W + w - O_i \right)$$
$$= w$$

it can be seen that Equation 30 is equal to the original analysis (Equation 28).
□

APPENDIX C: AN EXAMPLE TASK SET

This section applies the three tests (the original analysis, the exact offset test, and the Tractable Offset Test) to an example task set (this task set is originally due to Audsley [4]). The task set consists of 6 tasks (Table 2).

i	O_i	C_i	D_i	T_i	pri
1	4	1	1	10	1
2	5	1	2	10	4
3	0	5	6	20	2
4	7	8	9	40	3
5	27	8	14	40	6
6	0	6	30	40	5

Table 2: example task set

In the above diagram the column headed 'O_i' refers to the offset between the task and a notional time zero. The column headed '*pri*' denotes the priority of the task, with priority 1 being the highest priority[13]. All the above tasks are strictly periodic; tasks with the same period maintain the same relative offset relationship. One way of analysing the tasks would be to group all tasks sharing the same period into a single transaction, obtaining three transactions, of periods 10, 20, and 40. If this is done the following results are obtained:

i	r_i	D_i
A	1	1
B	10	2
C	6	6
D	15	9
E	40	14
F	30	30

Clearly, most tasks miss their deadlines. An alternative way of structuring the problem is to incorporate all the tasks into a single transaction of period 40. Tasks with period 10 are given four instances in the transaction; tasks with period 20 are given two. The following table shows the resulting transaction.

[13]This priority ordering is given by Audsley, and found according to the 'optimal priority assignment' algorithm [3]

i	A_a	A_b	A_c	A_d	B_a	B_b	B_c	B_d	C_a	C_b	D	E	F
pri	1	2	3	4	8	9	10	11	5	6	7	13	12
O_i	4	14	24	34	5	15	25	35	0	20	7	27	0
C_i	1	1	1	1	1	1	1	1	5	5	8	8	6
D_i	1	1	1	1	2	2	2	2	6	6	9	14	30
r_i	1	1	1	1	2	2	2	1	6	6	9	13	30

All deadlines for all tasks are met. Task *B* for the first three invocations just meets its deadline. For the final invocation it meets its deadline one tick early; task *E* meets its deadline one tick early. We reproduce below the STRESS diagram showing the execution of the above task set.

APPENDIX D: STRESS DIAGRAM NOTATION

In STRESS diagrams, time increases across from left to right. Task execution is represented by boxes. A task which is preempted is shown by a line at the level of the bottom of the boxes; a task which is blocked by a line at the level of the top of the boxes. These states are annotated by a variety of symbols:

Task release is marked by a low-level circle, and successful task completion by a high-level circle. If a task fails to meet its deadline, or otherwise fails to complete, then a filled high-level circle is used. Task deadlines are marked by a vertical line with an inverted 'v' mark at the bottom. Message transmission is marked by high-level outgoing arrow, and message reception by a low-level incoming arrow. An example is shown below.